THE LAYERED AGENT PATTERNS

Elizabeth A. Kendall, Margaret T. Malkoun
Computer Systems Engineering, Royal Melbourne Institute of Technology
City Campus, GPO Box 2476V, Melbourne, VIC 3001 AUSTRALIA
email: {kendall,maggie}@rmit.edu.au

OVERVIEW

This paper presents a collection of patterns as a starting point for a pattern language for agent based systems. After motivation and background, Section 2 discusses the context shared by all of the patterns. Section 3 describes the major architectural pattern, the ***Layered Agent***, while Sections 4 and 5 discuss the main patterns found within the layers, and relationships between the patterns, respectively. The paper concludes with key known uses, consequences, and a problem/solution summary for all of the patterns.

1.0 MOTIVATION AND BACKGROUND

Agent- based systems arise out of the following needs:

- *Adaptable, fault tolerant distributed systems that solve complex problems*
- *Open systems where components come and go and new components are continually added.*
- *Migration and load balancing across platforms, throughout a network.*
- *New metaphors, such as negotiation, for solving distributed, multi- disciplinary problems.*

A weak agent [22] is i) autonomous ii) social iii) reactive and iv) pro- active, and a strong agent, in addition to the these features, has one or more of the following: v) mentalistic notions (beliefs, goals, plans, and intentions) vi) rationality vii) veracity and viii) adaptability. An agent can migrate, and a multiagent system is open. Agents appear in a wide range of applications, including personalized user interfaces, enterprise integration, manufacturing, and business process support. Weak agents are assistants or drones [22].

It is the combination of autonomous, social, reactive, and pro- active behavior that distinguishes an agent from objects, actors [1], and robots. An agent is an object that is able to reason and to collaborate via structured messages. Agents encompass the behavior of Active Objects or actors, in that they have their own thread of control. The following examples illustrate the kinds of problems that agents address.

- *The air- traffic control systems in the country of ABC suddenly fail, due to weather conditions. Fortunately, agent- based air- traffic control systems in neighboring countries negotiate between themselves to deal with affected flights, and the potentially disastrous situation passes without incident.*
- *Upon logging into your computer, you are presented with a list of news group items, sorted into order of importance by your agent- based personal digital assistant (PDA). The assistant draws your attention to one article, which describes new work in your area. After discussion with other PDAs, yours obtains a relevant technical report for you from an FTP site. When a paper you have submitted to an important conference is accepted, your PDA makes travel arrangements by consulting a number of networked information sources.*
- *The new home robot developed by Company XYZ is engineered by a team of agent designers. Five disciplines --- marketing, mechanics, electronics, computers, and manufacturing --- are represented on the team, and the agents work together, negotiating with each other to develop a sound, concurrently engineered product.*

2.0 CONTEXT

All of the Layered Agent patterns presented in this paper share the same context that is based upon the following model of agent behavior [13]. Strong agents reason about their models of the world and themselves to select a capability or plan that could achieve their stated goal(s). A plan from the capabilities library is instantiated by the interpreter when a triggering event occurs, and an instantiated plan is an intention. An intention executes in its own thread, and an agent may have several intentions executing concurrently. Agents negotiate with each other in conversations or protocols. Agents coordinate their activities through market based economies, competitive bidding [14], and coalition formation [18]; the coalitions may have centralized or decentralized management. Strong agent collaboration across disciplines often requires the use of ontologies [20] so that cross disciplinary semantics can be exchanged.

These aspects of components of agent behavior are summarized in Figure 1 [13]. The agent's reasoning is within its declarative knowledge or models, interpreter, and capabilities library. The agent's knowledge is affected by sensors that monitor external objects. A capability is chosen by the interpreter and manifested as an intention; three sample intentions are shown. One intention involves an effector object that impacts external objects. The other two intentions feature collaboration with other agents, either within the original society or external to it. If the other agents are in another society, the agents must migrate, either virtually or in reality, in order to collaborate.

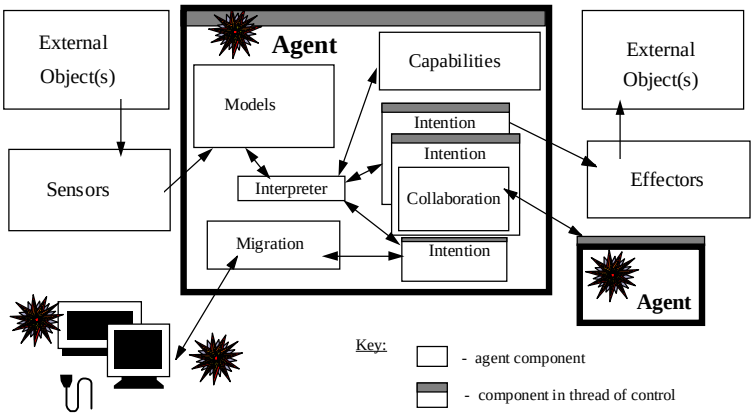

Figure 1: A Model of Agent Behavior

3.0 PATTERN: LAYERED AGENT

Problem:

How can agent behavior be best organized and structured into software? What software architecture best supports the behavior of agents ?

Forces:

- An agent system is complex, as illustrated in Figure 1.
- An agent is a large system that spans several levels of abstraction.
- There are many dependencies between the levels as shown in the diagram. These dependencies are between neighboring levels, and they feature two way information flow.
- The software architecture must encompass all aspects of agency
- The architecture must be flexible so both simple and sophisticated agent behavior can be addressed.

Solution:

Agents should be decomposed into layers [5] because i) higher level or more sophisticated behavior depends on lower level capabilities, ii) layers only depend on their neighbors, and iii) there is two way information flow between neighboring layers. For example, an agent can only collaborate if it knows what its own goals and capabilities are and if it knows how to communicate with other agents. Additionally, collaboration, with or without migration, exhibits two way information flow, with an agent functioning as either a client or a server.

The layers can be identified from the model of the agent's real world that is shown in Fig. 1. Each layer must be a reflection of one of the real world domains, and the domains must be ordered on the basis of their purpose in the system. Figure 2 structures Figure 1 into seven layers as follows:

1. *Sensory:* the agent gathers sensory input from its environment.
2. *Beliefs:* the agent's models of itself and its environment.
3. *Reasoning:* the agent determines what to do next, based on its goals, plans, capabilities, and models. The plans may generically specify requests for services or responses to requests.
4. *Action:* the agent's intentions (instantiated plans from the reasoning layer) are carried out. Pending actions for the agent are queued here, and these are derived from incoming requests that have been accepted by the collaboration layer.
5. *Collaboration:* the agent determines how to collaborate with another agent. This includes accepting or rejecting an incoming request for services. This layer addresses coordination protocols, including competitive bidding and coalition formation.
6. *Translation:* the agent formulates a message for another agent, translating it into another language or semantics (ontology), if necessary.
7. *Mobility:* when the agents who are collaborating with each other are in different locations, this layer is required for message transmission and reception.

In Figure 2, top down progress and information flow through the layers is shown on the left hand side of the figure, while bottom- up is on the right side. To summarize the bottom- up transactions, an agent's beliefs or models of itself and its environment are based on sensory input. When presented with a problem or a situation, an agent reasons about its beliefs to determine what to do. When the agent decides on an action, it can carry it out directly if it is within its own capabilities, such as sorting news items, but an action that involves other agents, such as concurrent engineering, requires the collaboration layer. Within this layer, the agent decides how to negotiate with another agent. Once the approach to collaboration is determined, the actual message is formulated at the translation layer if the agents speak different languages or come from different disciplines. Messages to distant societies are delivered by the mobility layer.

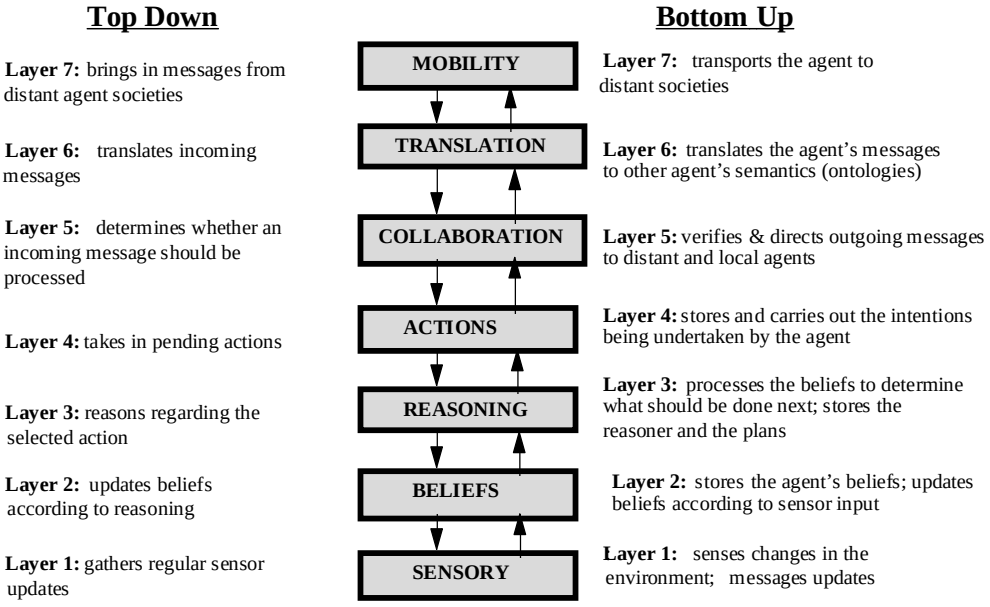

Figure 2: The ***Layered Agent*** Pattern

Top- down, distant messages arrive at layer 7, mobility. An incoming message is translated into the agent's dialect and semantics via the translation layer (if necessary). The collaboration layer stores contacts and conversations, and determines whether or not the agent should acknowledge and process a message. If the received message should not be processed, this reply is sent. If the message should be processed, it is passed on to the actions layer, where all pending actions are stored. When the action is selected for processing, it is passed to the reasoning layer. For example, the action may be a request from another agent for some information in the resident agent's beliefs. In this case, the action is processed within the reasoning layer, on the basis of a response from the beliefs in layer 2. The response is passed back to the actions layer. The lowest layer is layer 1, sensory, where the agent accumulates sensory information that updates its beliefs; the agent may also call on its sensors in response to an incoming request.

Once a plan is instantiated into an intention and placed in the actions layer, it does not require the services of any lower layers. An intention executes with the beliefs or models that the agent had when it (the intention) was instantiated. An intention does not get any new tasks to perform from higher layers, but it does call on the services of collaboration, translation, and mobility when it needs to involve other agents in its tasks. Intentions request these services synchronously, awaiting the replies.

4.0 PATTERNS INTERNAL TO THE LAYERS

In Section 4, commonly known patterns, such as ***Adapter, Mediator, Broker, Strategy,*** and ***Observer*** are only briefly described, while the ***Active Object, Interpreter***, ***Reasoner*** and ***Negotiator*** patterns are described in more detail.

4.1 Pattern: Active Object - Synchronization and Concurrency

Problem:

How do you manage different threads of control for agents?

Forces:

- Agents act in different threads of control and message each other with requests for services.
- An agent is a shared resource, in that many other agents may seek to collaborate with it.

- An agent must be able to react synchronously to messages at various priorities that have been sent by other agents.
- An agent may have various interface, implementation and synchronization policies that are not dependent on one another.
- Operating system behavior for thread management must be encapsulated.

Solution:

Figure 3 is an OMT diagram based upon the ***Active Object*** pattern [16] and the ACE toolkit [19] that address synchronization and concurrency. Four objects appear in the ***Active Object*** pattern: the Client Interface, the Scheduler, the Resource, and the Activation Queue. The Client Interface provides the interface to all clients. The invocation of a method triggers the construction of a Method Object which is then enqueued on the Activation Queue by the Scheduler. When the Resource is available, the Scheduler removes an activity from the queue and dispatches it to the Resource. Tasks are prioritized by the Scheduler when they are placed on the queue.

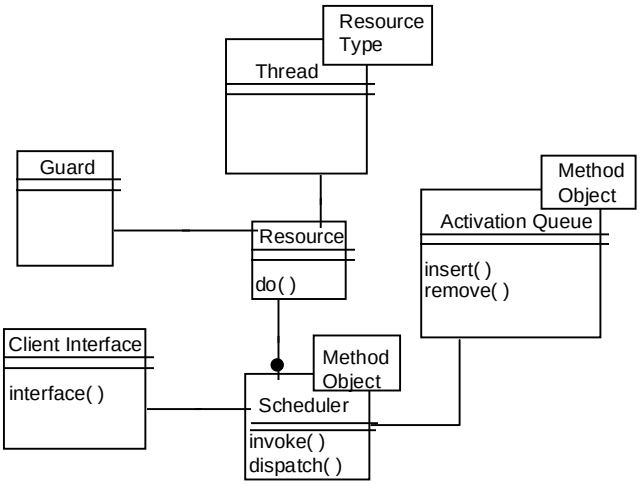

Figure 3: The ***Active Object*** Pattern with Thread and Guard Extensions

The additional two objects in Figure 3 are the Thread and the Guard. The Thread encapsulates operating system behavior for thread creation, termination, and management. Thread is a parameterized class, by Resource Type, and the Resource's do() member function is executed in the Thread. The Guard is responsible for managing exclusive access to the resource, when state must be preserved throughout a synchronized interaction. The Client Interface is responsible for non- exclusive synchronous interactions.

In the ***Active Object*** pattern, the Client Interface and the Guard decouple the synchronization behavior from the Active Object (Resource) itself, therefore separating and modularizing interface, implementation, and synchronization. This can be used to remove the inheritance anomaly. Whereas other patterns address some of the forces found in the problem of agent synchronization and concurrency, the ***Active Object*** is the only pattern that addresses all of them.

4.2 Pattern: Adapter - Interaction

Problem:

How do you get an agent to interact with external, coexisting objects?

Forces:

- The interface of an external, coexisting object must be converted to an interface an agent expects.
- Sensors must recognize the interface of external objects.
- Effectors must isolate an agent from platform or domain details.
- Agents need to work in various domains, and with various external objects without substantial changes.

Solution:

The ***Adapter*** pattern [8] translates incoming requests into a form that is understood by a domain object. This is applicable to the effectors in Figure 1 that pass the agent's actions on to coexisting objects, with the agent as the client who sends messages to an Adapter. The Adapter translates the messages and passes them on to the Adaptee. A sensor can be thought of as another form of an Adapter, addressing the fact that the agents follow structured messaging while the objects do not. In the case of the sensors, the agent is the Adaptee. For effectors, the external object is the Adaptee.

4.3 Pattern: Mediator - Centralized Collaboration

Problem:

How is an agent able to freely collaborate with other agents without direct knowledge of their existence?

Forces:

- Each agent may not have knowledge of every other agent
- Proliferating interconnections and dependencies increase complexity, complicate maintenance, and reduce reusability

Solution:

The ***Mediator*** [8] pattern is made up of a Colleague class category and two Mediator classes: an abstract Mediator that defines an interface for communicating with colleagues, and a concrete class that implements the behavior required to coordinate the colleagues. Each Mediator has associated with it a multitude of Colleagues, both at the abstract and concrete levels. With a Mediator, agents do not have to have direct knowledge of one another for collaboration. A Mediator that coordinates colleagues in different threads must be designed and implemented according to the ***Active Object*** pattern.

4.4 Pattern: Negotiator - Decentralized Collaboration

Problem:

How can agents collaborate directly with one another?

Forces:

- An agent may not have a Mediator to represent it
- When a Mediator is unavailable each agent must communicate directly with other agents, support different agent interfaces, and maintain knowledge of every agent it wants to interact with.
- Agents collaborate with each other via structured messages.
- An agent must be able to recover conversations that it has been involved in so that endless loops do not occur.
- Bottlenecks encountered in a centralized architecture need to be avoided.

Solution:

A ***Proxy*** [8] controls access to the Real Subject; it can also provide a distinct interface for each language. Languages such as KQML, COOL [2] and AgentTalk [14] have been developed to support agent collaboration. Whereas KQML is a very simple protocol for information exchange, COOL and AgenTalk represent conversations or protocols with a fixed number of states. The KQML Proxy would support the KQML interface, while the COOL Proxy and the AgenTalk Proxy subscribe to the other interfaces. In addition to intercepting messages, the Proxy provides a layer of indirection and is responsible for determining if the message should be passed on to the agent (the Real Subject).

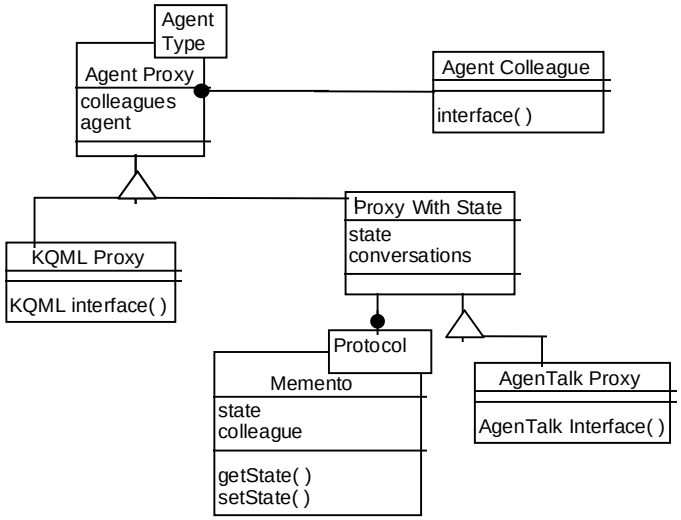

Figure 4: The ***Negotiator*** Pattern for Decentralized Agent Collaboration

For collaboration via coordination protocols, the agent must be able to determine its behavior based upon the state of the protocol [14] that it is engaged in. One agent may be engaged in several conversations simultaneously, requiring that it be able to accomplish context switching. The ***Memento*** pattern [8] externalizes an object's internal state so that the object can be restored to this state later. Agent proxies that support languages which involve conversations (such as the contract net) must store and recover their state. With the ***Memento*** pattern, they can delegate this to a companion Memento object. The Memento class is parameterized by Protocol, which gives the possible states and transitions. The resulting pattern for decentralized agent collaboration is shown in Figure 4.

Companion Memento objects are required for the coordination protocols that follow a conversation or a speech act finite state machine [14], and AgenTalk is shown in Figure 4. The Proxy is the originator of the Memento, and a Memento is instantiated when a conversation is undertaken with another agent colleague in the given language. The class Proxy with State therefore stores all conversations (instances of Memento) in a data member. When a collaboration is suspended, the Memento instance's state is stored so that the conversation can be subsequently resumed.

4.5 Pattern: Broker - Mobility

Problem:

How is an agent able to gain access to resources and other agents outside its society?

Forces:

- Agents must access each other and other resources across platforms.
- Making every agent responsible for access, security and interactions for a society leads to N- to- N connections and redundancy.

Solution:

The ***Broker*** pattern [5], which is an abstraction of Object Request Brokers, provides for communication and location transparency for objects that wish to be clients and servers of one another. With this pattern, an agent can be a client or a server to any registered agent. The agent (or its Proxy) can become a virtual member of open societies managed by the Brokers. Bridges between societies are also supported. Agents who wish to be clients and servers for one another must employ a Broker who is responsible for locating a server once a client has requested its services. Both the client and the server must register with the Broker. Also, the client and server employ proxies who respond to the interface known by the Broker; the proxies may also be located in a different (the Broker's) address space. The proxies shield an agent from a request that should not be answered. The ***Broker*** pattern provides virtual agent migration; actual migration has a substantial number of additional forces and therefore requires another solution.

4.6 Pattern: Reasoner - Automated Reasoning

Problem:

How is an agent able to select a capability or plan that could achieve its stated goal?

Forces:

- An agent can formulate appropriate reactions or response to stimuli.
- A plan is selected for instantiation when its invocation and context conditions are met.
- Invocation and context conditions are satisfied on the basis of the agent's beliefs or models of itself and its environment.
- A plan may have subplans.
- Automated reasoning may need to meet real time performance criteria.
- An agent's reasoning will need to be frequently updated or customized for new applications.

Solution:

An agent's automated reasoning capabilities have been designed as a forward-chaining, rule-based system. The rules are written in standard if <condition> then <action> format, and the agent determines its plan of action on the basis of its beliefs, through matching or satisfying the relevant invocation and context conditions. With this approach, each plan is a rule, with conditions that must be interpreted, and an action clause. The actions in the clause may involve effectors, collaboration, or mobility. Every action clause, when instantiated, becomes an intention. This provides the autonomous, reactive, and pro-active behavior that is central to the definition of an agent.

Two patterns used for agent automated reasoning are the ***Interpreter*** and ***Strategy*** patterns [8]. The ***Interpreter*** pattern defines a representation for the grammar and interpretation of a language. Interpreters relevant to agent oriented reasoning, that interprets beliefs, context conditions, and invocation conditions, have been designed via this pattern. The ***Strategy*** pattern encapsulates or objectifies an algorithm or a strategy for solving a problem. With this approach, each rule or plan can be an object, and this is shown in Figure 5. The key to the pattern is that each Plan has the same interface. Plan behavior involves instantiating intentions and checking conditions. Each Plan may consist of subplans. Using the ***Composite*** pattern, we can represent this hierarchy via specialized and composite plan classes (this is also shown in Fig. 5).

Figures 5 and 6 illustrate how the ***Interpreter*** pattern is used to represent the grammar and interpretation of the "if" clause (condition statements) of a Plan (rule). The Condition class declares an abstract interpret() operation. The grammar is represented by three subclasses, And/ Or (actually two separate subclasses, And and Or but shown as one here for brevity), Not and Variable. And/ Or and Not interpret condition statements with sub-conditions containing the logical operators AND, OR and NOT. A Plan creates and messages instances of Condition to interpret its condition statements; in this way, a Plan contains the Context and behaves as the Client in the ***Interpreter*** pattern [8]. And/Or and Not subclasses implement the interpret() operation recursively, breaking down the condition statement until a Variable (an atomic element) has been identified. If the Variable and the value of a Belief match, the condition of the plan is satisfied and the action clause of a plan would be instantiated into an intention.

Plans need to be examined when a triggering event, such as a change in the beliefs, occurs. Under these circumstances the Plans must continually check the Beliefs for any changes, hindering the performance of the Reasoning layer. To improve the performance, Beliefs can be the Subject in the ***Observer*** pattern. The Subject will then notify the Observers, one or more Plans, of any relevant changes that have occurred in the beliefs. This will remove the need to check beliefs that have not changed, and substantially improve performance. The relationship shown in Figure 5 between Beliefs and relevant Plans is two way --- a Belief knows which Plans depend on it, and a Plan knows which Beliefs are relevant to it.

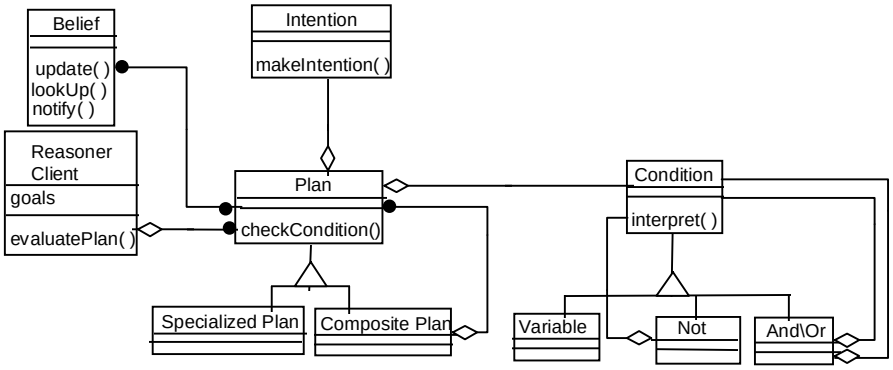

Figure 5: The ***Reasoner*** Pattern based on ***Interpreter*** and ***Strategy***

The Reasoner Client object shown in Figure 5 is needed for the top- down triggering of plans. A new goal comes in from the Actions layer to this object, which then passes it on to the relevant Plans, asking them to check their conditions. Goals are invocation conditions, whereas beliefs appear in context conditions. As such, a new goal leads to the same sequence of events as a new or changed belief; the plans check their conditions via the interpret() behavior of the Conditions.

Examples:

Collaboration between the classes in the ***Reasoner*** pattern can best be examined via the personal digital assistant (PDA), example and the event trace in Figure 6. In this example, upon logging into a computer, USERZ is presented with a list of news group items, sorted into order of importance by his PDA. The following plan, Plan Q, addresses this scenario:

```
IF      (NewItem, TRUE) AND
        (User, ON) AND
        (UserName, USERZ)
THEN
        Action: SortList
        Action: PresentSortedList
```

When UserZ logs in, the PDA senses this and messages Belief to update() with the facts (User,ON) and (UserName,USERZ). When a belief is updated, it notifies its associated plan(s), which then carry out checkConditions(). Plan creates an instance of an And Condition and then messages it to interpret() itself. As shown above, the condition statement for Plan Q is "NewItem is true and the user is logged on and the user is UserZ". Instances of the various Conditions classes are created and carry out their interpret() operations, breaking down the statement into sub-conditions and finally returning a variable to the Plan. The Plan checks its updated values for the Beliefs, and, if a match is found, then the condition is satisfied and the plan is applicable. The sequence 3: - 6: is repeated in Figure 6, due to the three clauses in the IF statement. The Plan then carries out makeIntention().

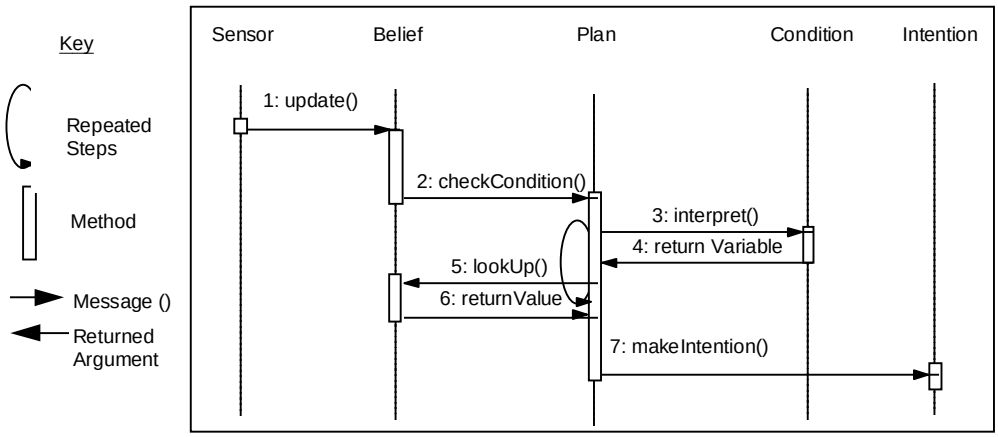

Figure 6: Collaboration between Sensor, Belief, Plans and Intention.

5.0 THE LAYERED AGENT PATTERN LANGUAGE

Now the internal or component patterns can be placed in their respective layers (Figure 7, 8 and 9) as a starting point for a pattern language for agent systems. Figure 7 indicates how the patterns are related, while Figure 8 indicates the relationships between particular components of the different patterns. The notation in Figure 8 indicates what pattern the component is from, matching that used in Figure 7.

5.1 Layers: Sensory, Beliefs, Reasoning, and Actions

The sensory layer is made up of Sensors, while the beliefs layer has Belief objects. As shown, the Sensors and the Effectors communicate with coexisting external objects. The reasoning layer is comprised of the objects from the ***Reasoner*** pattern, which select a Plan on the basis of Beliefs and goals, determining which plans should be intended. The mobility layer encapsulates the ***Broker*** pattern, with client and server proxies (for each agent or society), Brokers (shared), and bridges (shared). Except for the agent's own proxies, the components in the mobility layer are shared with and interact with other agents.

The actions layer has multiple purposes. First, intentions reside in this layer; they are not full Active Object***s***, but they do have their own thread of control. Asynchronous tasks from the intentions are carried out by the Effectors, which translate the agent's messages into commands for coexisting objects. Synchronous requests (collaboration with other agents) are output directly from the intention to the collaboration layer. Lastly, the agent's own Active Object components reside in the actions layer to handle incoming requests. That is, the agent's Client Interface, Scheduler, Resource, Guard, and Activation Queue are found here. If the agent

and the reply is forwarded via the Client Interface. If the request is asynchronous, the reply can be made via an intention.

5.2 Layers: Collaboration, Translation and Mobility

Agents share aspects of the mobility layer with other agents in a decentralized collaborative architecture (Figure 7). In a centralized collaborative architecture the translation and collaboration layer are shared with other agents in the same society and the mobility layer is shared between agent societies (Figure 9).

5.2.1 Decentralized Collaboration for Agents

The collaboration layer routes outgoing messages to a collaborator and translates them into an appropriate coordination protocol; it also determines whether or not an incoming request should be processed. This involves the Proxies with State and Mementos (for decentralized collaboration) in the ***Negotiator*** pattern, as shown in Figures 7 and 8. The upward links shown between Reasoner and Active Object (Figure 7) and Reasoner Client and Resource (Figure 8) are required for synchronous replies from the resident agent to another agent. Asynchronous replies are addressed by the Intentions. The Resource object shown in Figure 8 would pass control to the Reasoner Client, which may in turn pass control to the Beliefs and Sensory layers if sensors were required to look for updates to the Beliefs.

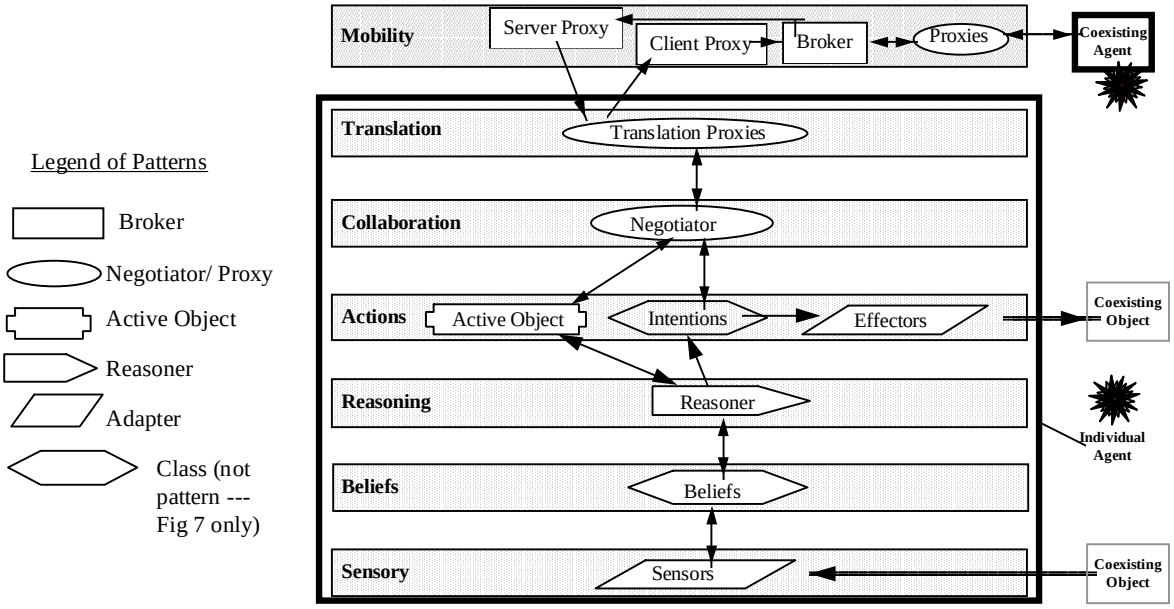

Figure 7: The Internal Layer Patterns and Their Interactions

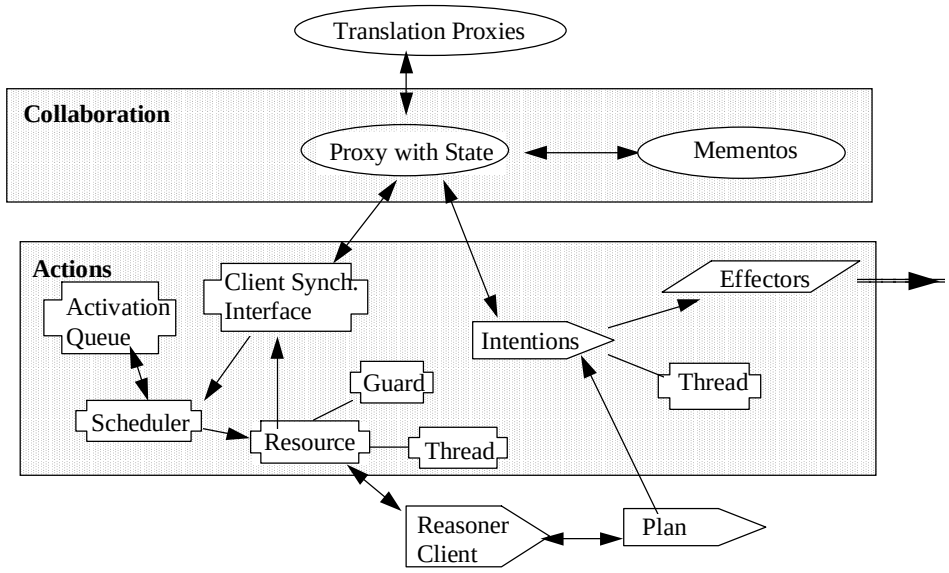

Figure 8: Interactions between Components in the Translation, Collaboration, Actions and Reasoning Layers

5.2.2 Centralized Collaboration for Agent Societies

A centralized collaboration layer is comprised of the Mediator(s) for a given agent society. The Mediator stores all contacts and conversations that are relevant to the aspect of the society that is under its guidance; it also has Translation Proxies which address translation and semantic information if it needs to communicate with another Mediator. The Mediator can deliver messages to agents within its own society. For messages that need to go outside of the society, the mobility layer is used. Centralized collaboration and translation are depicted in Figure 9. Two agent societies are shown, and they both have three agents (each with actions, reasoning, beliefs, and sensory layers) and centralized collaboration and translation layers. For cooperation across more than one society, the two societies share a mobility layer, with Client and Server proxies for the societies, along with the responsible Broker. (Any number of agent societies can share this mobility layer.)

The overall form or pattern language of the Layered Agent patterns can be seen in Figure 9. In the figure layers, patterns and classes appear. Agent societies are made up of any number of individuals, each with senses, beliefs, reasoning, and actions. With centralized organization, each society has Mediators or managers who delegate tasks and handle collaboration between agents who are under their supervision. If necessary, the Mediator calls on a translation layer for translation so that it can work with other Mediators or managers within its society. Societies then join mobility or network layers so that they can exchange services with other societies.

The difference between centralized and decentralized collaboration is that an agent is totally responsible for representing itself and carrying out its own negotiation in the decentralized case. In actual applications, the agents may work on their own for some considerations and rely on a Mediator for others, and this would lead to some shared collaboration layers being mixed in with some decentralized ones. The pattern in Figure 9 also addresses agent coalition formation [18] in that one or more agents may, during the course of their negotiation, decide to work together. They would then instantiate a Mediator and allow their collaboration layers to coalesce.

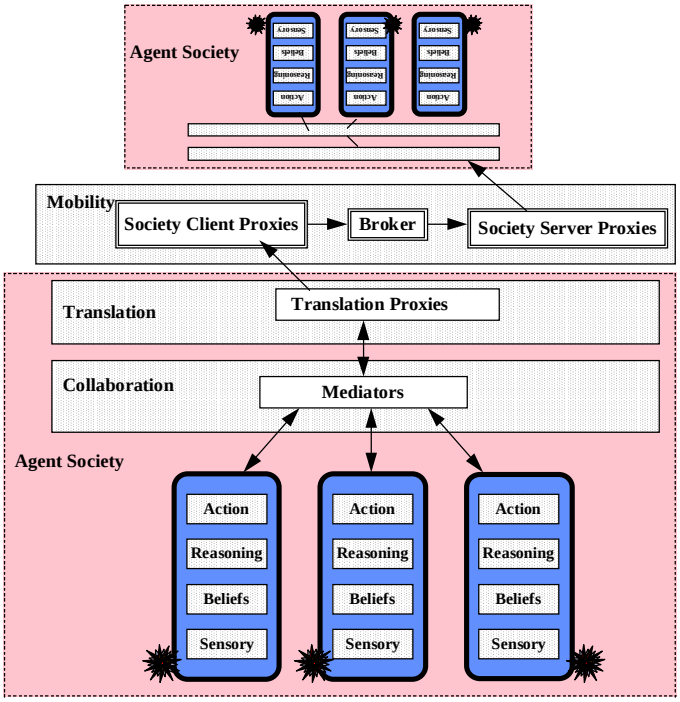

Figure 9: The Layered Agent Patterns with Centralized Collaboration and Translation

6.0 SUMMARY OF THE PATTERNS

6.1 Known Uses

The ***Layered Agent*** patterns can be found across the whole spectrum of strong agents. Strong agents include deliberative agents, planning agents, and agents with beliefs, desires, and intentions (a BDI agent) [22, 23]. A deliberative agent is able to sense, model, reason, and act; it therefore has the four lowest layers of agent behavior. Planning and BDI agents have more sophisticated actions layers, with intentions. Weak agents also have the four lowest layers, with reduced reasoning layers. By definition, all agents, weak and strong, have cooperative or collaborative behavior. If required, they also have translation and migration capabilities.

Layered Agent Architectures

Many examples can be found of layered agent architectures. A four layered architecture [16] was used in a traffic engineering application. This system consisted of a logical sensor layer - abstraction of concrete physical sensors, a skills layer - one or more specialized behaviours, a communication layer - decentralized communication protocol and an automata layer - planning system. The OSI network model was the foundation of many layered agent architectures, described in one application as Object Distributed Processing (ODP) agents [3]. The agents are organised as societies within the presentation and application layers of the OSI model.

GRATE [23] is a layered agent architecture for strong, cooperating agents; it features a domain level system, a cooperation layer, and a control layer. These are sensory, beliefs, reasoning, action and collaboration layers. TouringMachines [7] consists of perception, action and control subsystems. The control or reasoning subsystem mediates between the reactive, planning, and modelling layers. InterRRaP [17], another layered agent architecture, consists of four horizontal layers: cooperation, plan-based, behaviour-based and world interface. These correspond to the collaboration, action, reasoning, beliefs, and sensory layers. In InterRRaP, the horizontal layers are further subdivided vertically. Early architectures did not require mobility or translation layers.

Weak and Strong Agents

The ***Reasoner*** pattern, based on forward chaining rules, has been used in many knowledge based systems, including expert systems and agents. Particular agent based systems that use this form of reasoning are Homer [21], Prodigy [6] and ICARUS [15]. Additionally, these systems and PRS (Procedural Reasoning System) [9] parse or compile plans with a recursive interpreter, which is the vital aspect of the ***Reasoner*** pattern. For performance considerations, a dependency net (the ***Observer*** pattern) is used so that all beliefs do not have to be visited to check for changes. A plan with satisfied conditions becomes an intention.

In terms of the ***Reasoner*** pattern, a weak agent has a simplified reasoning layer, for example one that simply determines the appropriate action for a given stimulus or situation. This may mean the weak agent has only one plan. An example of this type of system is described in [11]. Sensors and effectors are added when weak and strong agents interact with coexisting applications [3] [21], such as databases.

The ***Active Object*** pattern has been used to implement actors [16], so the ***Active Object*** pattern components of the Layered Agent patterns are enough to constitute an actor, a common software module. An actor is comparable to a weak agent that does not work cooperatively and has one plan and one intention..

Some applications have relied on totally decentralized agent collaboration [12], whereas others have utilized coalitions [18] and various levels of centralized collaboration that are either static or dynamic. Proxies and mediators are well established solutions to decentralized and centralized collaboration, respectively. The known coordination protocols, such as KQML, COOL, contract net, and AgenTalk, are addressed by the proxies in decentralized collaboration. The translation layer proxies address ontologies [20].

Robots

Robots have the lower level capabilities of agents. Several layered architectures for robots exist [4]; the layers in the robots comprise senses, models, reasoning, and actions. There are two main approaches for building so-called ‘intelligent robots’. Traditional deliberative methods decompose the control problem into roughly sequential stages of perception, modelling, planning and execution (as found in the patterns discussed here). Behavioral approaches such as proposed by R. Brooks decompose the control problem into a parallel hierarchy of processes with ever increasing complexity of behavior. The agents layers discussed here would need to operate in both fashions, and not only sequentially, to address the behavioral approach.

Behavioral and deliberative robot systems that are made up of agents built from the Layered Agent patterns are now under development. The basic agent characteristics of autonomy, social interaction, reactive and pro-active response addressed by these patterns are useful as the stages in a conventional deliberative system, or as the layers in a behavioral architecture.

Agent Societies

The Layered Agent patterns provide a structure for modeling aspects of agents that reflect the behavior of humans and human societies. Individuals have senses, beliefs, reasoning, and actions. When presented with a problem or asked to accomplish a task, they reason about their models (usually not in a hierarchical fashion) to determine an action. When the humans interact with one another, they either do so directly or via Mediators or managers. Once a collaborator has been identified and an approach to collaboration has been formulated (either internally or through a Mediator), the message must be conveyed or communicated in the appropriate language. Lastly, societies or organizations deal with each other via Brokers, networks or other vehicles for exchange or mobility.

6.2 Consequences

The agent patterns described here have many of the benefits of any layered pattern [5]:

- *conceptual model* - the layers and their components address all of the key aspects of agency in a structured fashion
- *reusable building blocks* - individual layer implementations, for example sensors, can be replaced by other semantically equivalent ones
- *support for standardization* - for example, the CORBA standard can be used for the mobility layer; coordination and communication protocols can also be addressed by the layers
- *localized dependencies* - the synchronization behavior is localized to the Actions layer
- *changeability* - different collaboration layers can be brought in for centralized, decentralized, or hybrid collaboration. Each layer is only dependent on neighboring ones.

In addition, the patterns that are inside the layers have many demonstrated advantages. The ***Active Object*** pattern removes the inheritance anomaly. The ***Broker*** pattern and the ***Mediator*** pattern remove N-to-N connections and hence improve maintainability. Additionally, the pattern provided for decentralized collaboration delegates coordination and conversation storage, which enhances changeability and support for standarization within that pattern.

The major liability of these patterns would be performance, especially in the area of automated reasoning. An agent often needs to perform computationally intensive tasks that may not be best addressed by a layered architecture. However, the ***Observer*** pattern is a form of dependency network, and it plays a key role in the ***Reasoner*** pattern. This substantially improves the performance of the Reasoning layer.

The seven layer model also may be unneccessarily complex for many applications. However, as shown, the translation layer is only needed for communication across ontologies, and the collaboration and mobility layers may be shared with other agents. Additionally, the collaboration and mobility layers may coalesce when ***Mediator*** and ***Broker*** behavior can be combined.

Lastly, the ***Broker*** pattern discussed here for mobility only addresses virtual agent migration. Applications where fault tolerance and low network traffic are important, such as network management and WWW applications, would require actual agent migration. Truly mobile agents would require an enhanced mobility layer.

6.3 Problem/Solution Summary

Problem	Solution	Pattern
How can agent behavior be best organized and structured into software? What software architecture best supports the behavior of agents ?	Layered architectures have many benefits. Aspects of an agent are abstracted into a 7 layer model. Each layer provides a service to the layer above or below it. The layers are: a) mobility b) translation c) collaboration d) actions e) reasoning f) beliefs and g) sensory.	Layered Agent
How do you manage different threads of control for agents?	Decouple method execution from method invocation to simplify synchronized access to a shared resource by methods invoked in different threads of control. Removes inheritance anomaly.	Active Object
How do you get an agent to interact with external, coexisting objects?	Convert the interface of coexisting objects into an interface the agent expects. Adapter translates the messages and passes them on to either the agent or the effector from the sensor or agent respectively.	Adapter
How is an agent able to freely collaboration with other agents in a society without knowledge of their existence?	Encapsulate agent interaction in a Mediator object. The Mediator defines an interface for communication and coordinates agents so they do not have to have direct knowledge of the existence of one another.	Mediator
How can agents to collaborate directly with one another?	For direct collaboration each agent must support different interfaces and maintain knowledge of every agent it wants to interact with. Proxies can be used to provide distinct interface to languages and provide controlled access to the agent.	Negotiator
How is an agent able to gain access to resources and other agents outside its society?	Communication and location transparency is provided by a Broker. Agent can become a virtual member of an open societies managed by Brokers. Proxies must be employed for the client and server to able to respond to the interface of the Broker.	Broker

How is an agent able to select a capability or plan that could achieve its stated goal?	Using a rule-based approach the Interpreter processes the beliefs along with the Strategy (plan) to choose the most appropriate plan in order to achieve its stated goal.	Reasoner

7.0 ACKNOWLEDGMENTS

The authors wish to express their gratitude to Dirk Riehle for his expert advice as the shepherd for this paper. This project was partly funded by the Cooperative Research Centers for Intelligent Decision Systems and for Intelligent Manufacturing Systems and Technologies under the Australian Government's CRC Program.

8.0 REFERENCES

1. Agha, G., *A Model of Concurrent Computation in Distributed Systems*. 1986: MIT Press.
2. Barbuceanu, M. and M.S. Fox, "COOL: A Language for Describing Coordination in Multi-Agent Systems", First International Conference on Multi-Agent Systems, 1995.
3. Bernus, P., "Intelligent Systems Interconnection - Beyond OSI or Beyond ODP?", Open Distributed Processing Workshop, Sydney, 1990.
4. Brooks, R.A., "A Robust Layered Control System for Mobile Robot", IEEE Journal of Robotics and Automation, 1986. RA-2(1).
5. Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*. 1996: Wiley and Sons.
6. Carbonell, J.G., C.A. Knoblock, and S. Minton, *Prodigy: An integrated architecture for planning and learning*, in *Architectures for Intelligence*, K. VanLehn, Editor. 1991, Lawrence Erlbaum Associates: Hillsdale, NJ. p. 241-278.
7. Ferguson, I.A., "Towards an architecture for adaptive, rational, mobile agents", Proceedings of the Third European Workshop on Modelling Autonomous Agents and Multi-Agent Worlds (MAAMAW-91), 1991.
8. Gamma, E.R., R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. 1994: Addison-Wesley.
9. Georgeff, M.P. and A.L. Lansky, "Reactive reasoning and planning", Proceedings of the Sixth National Conference on Artificial Intelligence, Seattle, WA, 1993.
10. Huhn, A. and B. Wild, *Architecture for Distributed Agents*, in *Dependability of Artificial Intelligence Systems*, G.H. Schildt and J. Retti, Editors. 1991, Elsevier Science.
11. Ishida, T., "Parallel, Distributed and Multi-Agent Production Systems - A Research Foundation for Distributed Artificial Intelligence", International Conference on Multi-Agent Systems, 1995.
12. Jennings, N.R., P. Faratin, M. Johnson, P. O'Brien, and M. Wiegand, "Using Intelligent Agents to Manage Business Processes", First International Conference on the Practical Application of Intelligent Agents and Multiagent Technology, London, 1996.
13. Kendall, E.A., M.T. Malkoun, and C.H. Jiang, "A Methodology for Developing Agent Based Systems for Enterprise Integration", EI '95, IFIP TC5 SIG Working Conference on Models and Methodologies for Enterprise Integration, Heron Island, Australia, 1995.
14. Kuwabara, K., T. Ishida, and N. Osata, "AgenTalk: Describing Multiagent Coordination Protocols with Inheritance", submitted to Tools for Artificial Intelligence, 1995.
15. Langley, P., K.B. McKusick, J.A. Allen, W.F. Iba, and K. Thompson, "A Design for the ICARUS Architecture", SIGART Bulletin 2, 1991: p. 104-109.
16. Lavender, R.G. and D.C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming", Pattern Languages of Programming, Illinois, 1995.
17. Muller, J.P., M. Pischel, and M. Thiel, "Modelling interacting agents in dynamic environments", Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI-94), Amsterdam, 1994.
18. Sandholm, T.W. and V.R. Lesser, "Issues in Automated Negotiation and Electronic Commerce: Extending the Contract Net Framework", First International Conference on Multiagent Systems, 1995.
19. Schmidt, D.C., "The ACE Object-Oriented Encapsulation of Light Weight Concurrency Mechanisms", 1995.
20. Tenenbaum, J.M., J.C. Weber, and T.R. Gruber, *Enterprise Integration: Lessons from SHADE and PACT*, in *Enterprise Integration Modeling Proceedings of the First International Conference*, C.J. Petrie, Editor. 1992, MIT Press.
21. Vere, S. and T. Bickmore, "A Basic Agent", Computational Intelligence, 1990(6): p. 41-60.
22. Wooldridge, M.J. and N.R. Jennings, "Agent Theories, Architectures and Languages", ECAI-94 Workshop on Agent Theories, Architectures, and Languages, Amsterdam, 1995.
23. Wooldrige, M.J. and N.R. Jennings, *Intelligent Agents ECAI-94 Workshop on Agent Theories, Architectures, and Languages*. Lecture Notes in Artificial Intelligence, ed. J.G. Carbonell and J. Siekmann. 1994: Springer-Verlag.